Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Усов Илья Дмитриевич

Повышение производительности алгоритмов распределенного кэширования

Курсовая работа

Научный руководитель:
ст. преп. С. Ю. Сартасов

Санкт-Петербург
2017

Оглавление

Введение

На сегодняшний день ни один Web-проект не сможет обойтись без кэширования. Кэширование — это один из наиболее эффективных методов повышения производительности современных систем. За счет хранения части данных в более быстрой памяти, чем первоначальный источник, и достигается улучшение. Успех такого подхода кроется в принципе локальности (в определенный промежуток времени происходит обращение только к какому-то подмножеству данных). Как правило, размер кэша сильно ограничен, следовательно алгоритмы кэширования должны решать, какие элементы должны храниться, а какие нет. Естественно, что решение должно приниматься эффективно, то есть время на принятие решения не должно перекрывать преимуществ использования кэша. Однако при высоких нагрузках может случиться, что сервер будет просто не в состоянии быстро обработать все запросы. Из-за этого увеличивается время отклика, что приводит к деградации сервиса. В таких случаях нагрузку распределяют сразу на несколько серверов (кластер). Сервера, входящие в кластер, могут иметь различные объемы памяти, однако общий объем кэша равен сумме всех кэшей. Такой подход и называется распределенным кэшированием.

1. Постановка задачи

Цель работы - сделать обзор производительность существующих алгоритмов распределенного кэширования и реализовать наиболее подходящий алгоритм в рамках библиотеки Redis[7]. Реализация должна быть написана на языке C. В качестве тестовых данных для сравнения производительности будут использоваться запросы, распределенные по *hotspot*-распределению.

Данную работу можно разбить на несколько подзадач:

- Сделать обзор существующих алгоритмов кэширования.
- Сделать обзор существующих решений в области распределенного кэширования.
- На основе полученных данных реализовать эффективный алгоритм кэширования в рамках библиотеки с открытым исходным кодом Redis[7].
- Выбрать критерии и провести сравнение своего алгоритма с уже существующими.

2. Обзор

2.1. Алгоритмы вытеснения

2.1.1. Least recently used (LRU)

LRU[5] - одна из наиболее популярных политик вытеснения, которая вытесняет элемент, который дольше всего не использовался. Данный алгоритм прост в реализации (в связном списке, элемент, к которому произошло обращение передвигается в голову списка) и имеет высокий процент случаев, когда данные были взяты из кэша, а не из основного хранилища данных. Но он не учитывает ситуации, когда к некоторым элементам обращение производится довольно часто, однако они успевают покинуть кэш. Примером такой ситуации может служить "полное сканирование", когда на протяжении некоторого промежутка времени обращения происходят преимущественно к новым элементам.

2.1.2. Least frequently used (LFU)

LFU[1] - политика, выталкивающая элемент, который используется наименее часто. Каждому элементу кэша сопоставляется так называемый счетчик обращений. При попадании элемента в кэш, счетчику устанавливается значение 1, при последующих обращениях к элементу, счетчик увеличивается на 1. Данная политика не лишена существенных недостатков. Элемент, к которому за короткое время было много обращений надолго зависает в кэше. Алгоритм никак не учитывает возраст страницы.

2.1.3. Low inter-reference recency set (LIRS)

LIRS[3] - политика для каждого элемента X, оценивает количество элементов, обращение к которым производилось между двумя последними обращениями к X (IRR(X) - *Inter-Reference Recency*). А так же количество уникальных обращений к элементам, после последнего обращения к X (R(X)). Идея алгоритма строится на том, что если IRR

высок, то ,скорее всего, и дальше будет высоким. Все элементы разделяются на LIR (*Low* IRR) и HIR (*High* IRR). В случае, если надо вытеснить элемент, выбирается один из HIR. Каждый новый элемент N кэша сравнивается со всеми LIR элементами и в случае, если находится элемент, который хуже чем N, то он становится HIR, а N - LIR. Данный алгоритм довольно эффективен, однако сложен в реализации и требует пересчета IRR и R для каждого элемента при каждом запросе.

2.1.4. Multiple queue (MQ)

MQ[9] - политика использует несколько списков элементов, находящихся в кэше. Каждый список построен по принципу LRU. Каждый элемент имеет свой счетчик (аналогично LFU). MQ есть список элементов, которых сейчас нет в кэше, однако они были там относительно недавно (буфер истории). Для элементов находящихся в буфере, счетчик также сохраняется. Номер списка, в котором должен находится элемент, вычисляется в зависимости от счетчика.

При обращении к элементу возможны 3 варианта:

- Искомый элемент находится в кэше. Тогда он перемещается в начало соответствующего списка(это может быть список, в котором элемент уже находится).
- Искомый элемент находится в буфере истории. Тогда элемент перемещается в соответствующий список, а из него в свою очередь вытесняется лишний элемент и помещается в буфер.
- Искомого элемента нет ни в кэше, ни в буфере истории. Тогда один из элементов кэша перемещается в буфер истории, а из него в свою очередь удаляется одна запись. Искомый элемент помещается в кэш.

Для того, чтобы в кэше не висели ранее популярные элементы в MQ для каждого элемента введено понятие expiredTime = currentTime + lifeTime(параметр системы). При каждом запросе проверяется условие

currentTime > expiredTime и если оно выполняется, то элемент перемещается в начало нижележащего списка(если ниже спускаться некуда, то перемещается в конец текущего списка).

2.2. Существующие библиотеки распределенного кэширования

2.2.1. Memcached

Memcached[2] — одна из самых популярных библиотек распределенного кэширования. По сути Memcached — хеш-таблица хранящаяся в оперативной памяти, доступ к которой осуществляется по сетевому протоколу. Таким образом обеспечивается сервис по хранению данных, доступных по ключам. Ключом является строка ограниченной длины, с ограниченным набором символом. Memcached спроектирован таким образом, что все его операции имеют алгоритмическую сложность O(1). Таким образом все операции выполняются очень быстро, однако из-за этого некоторые операции отсутствуют в нем (нет групповых операций над ключами или их значениями и т.п.). Например, отсутствуют возможность объединять ключи в группы. Memcached использует LRU политику вытеснения (O(1) достигается за счет хэширования). Используется асинхронный ввод-вывод. Нити (*threads*) используются только в случае слишком большой нагрузки, чтобы задействовать все доступные ядра или процессоры на сервере. В кластере все ключи равномерно распределяются по серверам. Основная суть алгоритма распределения: рассматривается набор чисел из кольца классов вычетов по модулю 2^{32}. Каждому серверу сопоставляется число из кольца (точка сервера). Каждому ключу сопоставляется число в том же диапазоне (точка ключа). Сервер, на котором будет хранится ключ, определяется ближайшей точкой сервера к точке ключа по возрастанию. При отключении сервера из кластера или добавлении на кольце уберется или добавится точка сервера. Соответственно перераспределяться будет лишь часть ключей[10]. В случае, когда обращение происходит к серверу, который

не отвечает за данный ключ, запрос пересылается на соответствующий сервер. Memcached в основном используется для кэширования малых и статистических данных.

2.2.2. Redis

Redis[7] - еще одна популярная библиотека распределенного кэширования. Так же, как и Memcached, Redis представляет собой хэш-таблицу, хранящуюся в оперативной памяти. Важным отличием является то, что необязательно строка может быть ключом. В Redis реализовано пять структур данных: Строки (*strings*), хэши (*hashes*), списки (*lists*), множества (*sets*) и упорядоченые множества (*sorted sets*)[8].
При достижении максимального ограничения объема памяти, Redis действует в соответствии с одним из следующих правил (правило указывается в конфигурации):

- **noeviction**: Возвращается ошибка, если достигнут предел памяти и пользователь пытается выполнить операцию, которая может привести к еще большему количеству данных
- **allkeys-lru**: Вытесняются ключи, согласно политике LRU
- **volatile-lru**: Вытесняются ключи, согласно политике LRU, но только те, которые имеют срок жизни(TTL - *time to live*)
- **allkeys-lfu**: Вытесняются ключи, согласно политике LFU
- **volatile-lfu**: Вытесняются ключи, согласно политике LFU, но только те, которые имеют срок жизни(TTL - *time to live*)
- **allkeys-random**: Вытесняются случайные ключи
- **volatile-random**: Вытесняются случайные ключи, но только те, которые имеют TTL
- **volatile-ttl**: Вытесняются ключи с наименьшим TTL

Реализация LRU (Approximated LRU) в Redis не является точной реализацией LRU. Redis хранит отсортированный массив из 16 элементов, который обновляется каждый раз, когда необходимо освободить память. Обновление происходит следующим образом:

1. Берется n случайных элементов (n указывается в конфигурации)
2. Для каждого элемента считается характеристика (idle), относительно которой сортируется массив
3. Элементы вставляются в массив таким образом, чтобы массив оставался отсортированным

Сам массив не синхронизирован с основным хранилищем. На практике такой подход дает почти такой же результат, как и оригинальный LRU, однако занимает меньше памяти[4]. За распределенность в Redis отвечает Redis Cluster. Данные распределяются по кластеру с помощью хэш слотов (*hash slots*). Всего 16384 слота. Каждый узел в кластере отвечает за некоторое подмножество слотов. Например, на 3 сервера мы можем распределить слоты таким образов (0 — 5500, 5501 — 11000, 11001 — 16383). Ключу сопоставляется один из слотов, после чего ключ отправляется на соответствующий сервер. Если происходит обращение к узлу, который не отвечает за этот ключ, происходит перенаправление на правильный узел. Redis Cluster предоставляет возможность перемещать слоты от одного узла к другому. Благодаря этому можно добавлять и удалять узлы во время работы кластера. Redis поддерживается master-slave репликацию. В автономном режиме Redis позволяет переключаться между базами данных. Redis Cluster не поддерживает несколько баз данных.

2.2.3. Hazelcast

Hazelcast - библиотека распределенного кэширования. Поддерживает LRU и LFU политики вытеснения.
Для каждой записи определяются следующие свойства:

- **time-to-live**: Максимальное время в секундах нахождения записи в карте. Если оно не равно 0, то записи, которые не обновлялись дольше этого времени, вытесняются из памяти автоматически.

- **max-idle-seconds**: Максимальное время в секундах бездействия записи в карте. Если оно не равно 0, то записи, которые бездействовали дольше этого времени вытесняются из памяти.

Для каждой карты определяются следующие свойства:

- **max-size**: Максимальный размер карты. Когда максимальный размер достигается, происходит вытеснение лишних элементов согласно выбранной политике.

- **eviction-percentage**: Процент записей, которые необходимо вытеснить из памяти при достиженнии max-size.

В Hazelcast реализованы следующие распределенные структуры: карты (*maps*), очереди (*queues*), множества (*sets*), списки (*lists*), карты с несколькими значениями для одного ключа (*multimaps*), карты копирующие данные на все узлы кластера (*ReplicatedMap*).

3. Описание решения

3.1. Алгоритм кэширования

Прежде всего, стоит заметить, что в Redis решение о вытеснении каждый ведущий узел принимает самостоятельно. Без учета статистики ведомых и других ведущих. Так как каждый ведущий отвечает за свой набор ключей, который не пересекается с другими наборами, можно рассматривать только одного ведущего и его подчиненных. Поскольку ведомые можно использовать для масштабирования чтения, было бы неплохо при выборе кандидата на выселение учитывать и статистику обращений к элементам на ведомых. Иначе может получиться так, что ведущий будет вытеснять ключи, которые не популярны на нем, но популярны на ведомых.

3.1.1. Первый подход

Redis хранит пул кандидатов на выселение. Для того, чтобы снизить количество вытеснений элементов популярных на ведомых и непопулярных на ведущем, можно попытаться синхронизировать пул ведущего с пулами подчиненных. Для этого можно использовать следующий алгоритм

1. Раз в определенное время ведущий отправляет всем своим ведомым свой список кандидатов на выселение

2. Каждый ведомый в ответ отправляет обратно ведущему только те элементы этого списка, которые более популярны на ведомом, чем на ведущем

3. Раз в определенное время каждый ведомый отправляет ведущему свой собственный список кандидатов на выселение

4. Ведущий на основе полученных данных от ведомых обновляет свой пул

Таким образом в пуле ведущего будут находиться актуальные данные, полученные на основе статистики как самого ведущего, так и ведомых. Однако в ситуации, когда вытеснения происходят очень часто, пул будет обновляться очень часто и с большой вероятностью те обновленные данные, которые вернутся ведущему от ведомого, будут либо вытеснены из пула, либо вытеснены вообще. Из-за этого действия, направленные на проверку того, стоит ли вытеснять данные с ведущего могут, быть малоэффективны.

3.1.2. Второй подход

Вместо того, чтобы передавать данные с ведущего на ведомый и обратно в попытке синхронизировать пулы, можно попробовать собирать пул сразу на данных, полученных от ведомых:

1. Раз в определенное время ведомые присылают мастеру список кандидатов на выселение
2. Полученные данные ведущий сохраняет в буфер
3. Когда наступает необходимость обновить пул, в первую очередь пытаемся взять данные из буфера, а уже во вторую – случайные элементы

При таком подходе нам не надо постоянно обновлять пул. Однако данные в пуле не всегда будут актуальные.

3.2. Реализация

Было создано ответвление (fork) библиотеки с открытым исходным кодом Redis, в рамках которого были реализованы предложенные алгоритмы. Реализация написана на языке C.

3.2.1. Передача данных

Данные передаются между узлами с помощью Redis API. На узел назначения для каждого ключа отправляются следующие данные:

1. Название команды, по которой вызовется необходимый обработчик
2. Ключ
3. Значение, необходимое для подсчета характеристики (idle), относительно которой сортируются ключи в пуле (для LRU и LFU оно разное, однако в обоих случаях занимает 24 бита)
4. Номер базы данных (dbid)

Узел, получив эти данные, вызывает обработчик, привязанный к этой команде и передает ему полученные аргументы.

3.2.2. Буфер

Во втором подходе данные, полученные от ведомых, складываются в буфер, который представляет из себя массив из n элементов (по умолчанию 100). Каждый элемент представляет собой структуру данных, которая содержит:

1. Ключ
2. idle
3. dbid

В буфере хранятся копии ключей, чтобы не возникало проблем, когда ключ будет вытеснен из основного хранилища. Когда буфер полностью заполняется, он начинает заполняться с начала, при этом не очищая данные, которые уже в нем хранятся, до момента, пока их не перезапишут.

3.2.3. Обновление пула

В первом подходе пул обновляется в двух случаях:

1. Подчиненные устройства прислали своих кандидатов на выселение

2. Необходимо освободить память

Во втором подходе, сперва берутся элементы из буфера. Если их в нем недостаточно, то недостающие элементы выбираются случайно.

3.3. Апробация

Апробация данных алгоритмов производилась при помощи симулятора Redis[6], в который были добавлены различные распределения для сравнения. При апробации измерялись статистики на одном ведущем и двух ведомых устройствах, к каждому из которых было подключено по два клиента и бралось среднее. Сравнения производились на *hotspot*-распределении, то есть распределении при котором на x% данных приходится y% обращений. Ниже приведена точность различных алгоритмов для данного распределения (Redis LRU, Redis LFU и описанные выше алгоритмы).

Алгоритм	Точность (70/20)	Точность (75/25)	Точность (80/35)
Redis LRU	69.39%	69.93%	62.12%
Первый	70.47%	70.13%	61.75%
Второй	69.79%	69.85%	61.22%

Таблица 1: Точность различных алгоритмов кэширования на основе LRU.

Алгоритм	Точность (70/20)	Точность (75/25)	Точность (80/35)
Redis LFU	71.24%	68.92%	60.78%
Первый	69.25%	70.08%	61.38%
Второй	71.30%	69.31%	61.34%

Таблица 2: Точность различных алгоритмов кэширования на основе LFU.

Как следует из данных представленных выше, предложенные алгоритмы по точности сопоставимы с исходным алгоритмом.

Ниже приведена статистика количества запросов обрабатываемых каждым узлом в секунду.

Алгоритм	Ведущий	Ведомый 1	Ведомый 2
Redis LRU	14191	13016	12988
Первый	12616	10975	11016
Второй	14025	12825	12958

Таблица 3: Количество запросов в секунду к узлу при различных алгоритмах кэширования

На основании вышеизложенных данных, можно сделать вывод, что предложенные алгоритмы не дают прироста производительности.

Заключение

В ходе данной работы были достигнуты следующие результаты:

- Сделан обзор основных политик вытеснения.
- Сделан обзор существующих решений распределенного кэширования.
- Реализованы оба предложенных алгоритма в рамках библиотеки с открытым исходным кодом Redis
- Сравнена эффективность алгоритмов на различных данных

Исходный код данной работы можно получить по адресу `https://github.com/Iliya-usov/redis`.

Список литературы

[1] EFFELSBERG WOLFGANG, HAERDER THEO. Principles of Database Buffer Managements // ACM Transactions on Database System. — 1984.

[2] Interactive Danga. Memcached. — Access mode: `http://memcached.org`.

[3] Jiang Song, Zhang Xiaodong. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance // Department of Computer Science College of William and Mary Williamsburg. — 2002.

[4] Redis. Using Redis as an LRU cache. — Access mode: `https://redis.io/topics/lru-cache`.

[5] SACCO GIOVANNI MARIA, SCHKOLNICK MARIO. Buffer Management in Relational Database Systems // ACM Transactions on Database System. — 1986.

[6] Sanfilippo Salvatore. Performing simulation. — Access mode: `https://redis.io/topics/rediscli`.

[7] Sanfilippo Salvatore. Redis. — Access mode: `https://redis.io`.

[8] Seguin Karl. Little Redis Book. — Access mode: `https://github.com/karlseguin/the-little-redis-book`.

[9] Zhou Y., Philbin J. F. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches // Proceedings of the 2001 USENIX Annual Technical Conference. — 2001.

[10] Смирнов А. Web, кэширование и memcached. — 2008. — Access mode: `http://highload.guide/blog/web-caching-memcached.html`.